

Mythology, Psychology, Technology and Education

A spirit walk through computing

Most people think they can't really understand computers. They're too complicated, you're either "a computer person" or you're not. They can learn applications. They can't learn code-- it's too much like Math. Well, it's fundamentally Math, that much is true.

The best thing a teacher can do is teach someone how to learn. The traditional way of getting this wrong is to teach people to recite facts. Take a bunch of pieces of information, stick them in your head, and show them off whenever it's time to demonstrate what you've learned.

The terrible thing about this is how little regard it has for the way people learn. Instead of focusing on helping the student to retain knowledge, it simply gives them the task of retaining it and says "here, you work it out." Instead of being given an education, it is simply assigned to them.

After people get out of school, they have the opportunity to learn in a more natural way-- but they may already associate learning with school, and thus believe that learning is the obtaining and recital of facts. They know that application fits into the picture-- but it's sort of moot because application comes after learning.

Well, that's why education is akin to vandalism.

Application is part of learning, it doesn't come after. Sure, you don't want to take a student into the O.R. to perform a vital surgery. And you don't want to leave them stranded in a clinic where they are expected to perform first aid without someone experienced to act as a guide.

But some of the masters of their fields began as apprentices. And among teachers, mentors are the exception rather than the rule.

When that's how education is practised, It should follow that students who learn well are also the exception, rather than the rule.

So here we are on another page, and the first lesson will be how to learn. This is only an introduction, but it will help you learn about computers. If you think we are only going to talk about the imagination, consider this: science is more concerned with questions than facts. Answers lead only to better questions, and Facts are more of a by-product of science.

So if we have an educational system that's based on reciting facts, it is a very dogmatic, inflexible and unimaginative one. It should be no wonder then that we have so many dogmatic, inflexible and unimaginative people. They are not born as such-- but raised and taught to have preconceptions that often don't assist learning but actually create people who firmly believe they are incapable of grasping entire subjects of knowledge!

And why not-- after years of failing to grasp something that was never taught, why not assume it is the student's fault? After all, teachers can't fail (and it's still the student's job to learn either way, regardless of whether anybody teaches them how to learn.)

So, if you want to learn better than people are able to do in school, the first thing you want to do is fix your broken relationship with information.

Here is what school teaches-- you take facts, memorise them, and then prove you know them later. After that, you can leave school and use the information.

Here is how people naturally learn-- you examine the world, you make a mental note of your reactions to it, you ask questions about your reactions-- you re-examine the world in an effort to find answers, and then you compare the mental notes with what you've learned.

The nice thing about the second way is that it is like going for a walk in a cool breeze, compared to formal education's running a marathon on a hot summer day. The other nice thing about the second way is that it works the same way the brain does.

The brain doesn't store "facts" in a sequence the way a piece of paper or DVD does, or the way that a syllabus is typically designed. It recognises and integrates patterns. If you learn by looking for and recognising patterns, you are taking full advantage of your own nature.

If you learn in a strict sequence, without taking advantage of your nature, you are basically stomping your way through town. You will eventually get where you're going, but your legs and body (which also fuels your ability to think and learn) will be exhausted after a short time. Instead of wanting to learn, you'll learn to avoid it.

And that's what school does, it creates a visceral fear of learning what you don't already know. There is a concept in pedagogy called the "Zone of proximal development." It's a pretty obvious concept to confirm, but basically the stuff you are most capable of learning is the closest to what you already know.

There's the stuff you already know, the stuff you can learn from what you already know-- and the stuff you can't learn at this time.

As long as you stay inside the zone of what you can learn from what you already know, learning is easy and often fun. And any time you go outside the zone, it becomes difficult and tedious and inefficient.

We can then create a syllabus that tries to stay inside this zone, and force people to plod through it as quickly as possible. In theory, the ZPD will expand, the student will learn, the teacher will get paid for their work, and the school will get sustained funding.

The problem isn't the theory, it's the syllabus. Basically the zone is a 3-dimensional sphere moving outwards, and the syllabus is a straight line in one direction.

Now, maybe this leaves you feeling lost and you're ready to give up right here, or maybe you've heard it all before and haven't applied it. Because one thing you can take away from all this is that it's unscientific to assume you can't learn an entire subject. It's prejudice against yourself, it's a very superficial assumption.

To really know what you are incapable of learning, you have to try. But

you have to know that a large part of the reason people don't learn is that they don't know how. School doesn't teach people how to learn-- it actually teaches them how to make learning less possible and more difficult.

The reason this is called *Mythology, Psychology, Technology and Education*, is that it connects these ideas together.

A few years ago I wrote a short story about a hypnotist that can control robots with hypnosis. This is meant to be educational and illustrative, but it also comments on what people think about computers. People are used to speaking to each other (and their machines) on the surface, where their input immediately hits a bunch of "routines" that both people and machines have.

It's an old truism that computers are stupid and need to be told everything, and they don't "think" like we do.

And that's a useful lesson for someone trying to understand programming, but the more we move forward with AI (and we are there now, in fact we were right about a lot of old/original theories about AI that we thought were quaint and naive-- only because the equipment wasn't powerful enough to prove very much capability) but the truth is, that the more we learn about the brain and AI, the more they have in common.

At the very least, we can spend another few hundred years debating if "machines have souls too," but the comparison of computer to brain gets less silly with time and advances in technology. New networks are going online right now that simulate rodent brains. They look like modern versions of the old computer rooms of the 1960s-- or they look like server farms-- but their purpose is to simulate and teach us more about the brain itself.

The good news is, it's still easier to understand computers-- in part because we (as a species) designed the things.

As for hypnosis, there's a TED talk about whether hypnosis is real. <https://www.youtube.com/watch?v=1RA2Zy_IzfQ> In the talk, a stage hypnotist explains the history of hypnotism and hypnotises several people from the audience.

Outside of a TED talk, such a performance could be accomplished with actors mixed into the audience and chosen as "volunteers." This is also possible in a TED talk, though the hypnotist also talked about the confluence of hypnosis and science-- and explained why books about hypnosis are more likely to be found in the Occult section.

Before writing a short story that compared hypnotising robots to computer programming, I wanted to teach people "computer psychology"-- How to understand the way your computer thinks. The idea is that if people know how the computer thinks, that will make it much easier to understand how to communicate with it. If they understand how to communicate with it, they will know how to program it.

Just to give you an idea of what I mean in real terms, when I got the computer I am typing this on, I hadn't even left the store before I completely changed all the software running on it. First I plugged in a USB stick. Then I restarted the computer and hit a few keys. Just a few.

Then instead of running Windows, the computer was running other software from the USB-- it loaded, I took the USB stick out, then I was running a different Operating System on the computer. It still needed the USB stick to load it-- but not to run it.

So I hit a few more keys, and told it to erase everything on the computer. It's my computer, and I had no reason not to do that. Now instead of Windows, I told it to install this system on the computer so I wouldn't need the USB stick to start it again.

It's a process that often takes about 10-15 minutes. And the purpose is to have a working computer that I like a lot more than Windows.

But the point is, if you give me a computer I can demonstrate this on most models (my USB stick is designed for 32 and 64-bit x86-based CPUs, and that's most of the laptops you'll find, though some are ARM-based and others are Mipsel processors. I've owned both of those as well. Your phone has an ARM-based processor.)

So when I compare computing to mythology and psychology-- I admit, my understanding of psychology has limits. But trust me (at least for

the moment) on the computer part.

Psychology pioneers Freud and Jung built entire philosophies based on studying dreams, hypnosis and mythology. They used these as early tools for understanding how the brain works-- and you can use them as tools for understanding and communicating with the computer. But the main reason I brought up mythology was not to talk about Jung's theories about archetypes, but to demonstrate both the costs of not understanding computers and a way of making learning about computers more effective.

This is getting back to what I said about education, and how instead of exploring the expanding sphere outside what you immediately already know-- it drags the student in a straight line outward from their established knowledge towards the subject matter. And this is unnatural and inefficient and tedious.

When it works, it misleads people into thinking they understand something and when it fails, it makes people believe (I stress the word "believe" because it is a false belief and yet people don't question it enough) that they are incapable of learning entire subjects. If we are really going to make people capable of learning subjects they believe they can't learn, it pays to address the belief-- they may find there are other things they falsely believe they can't learn.

Getting back to mythology, it is useful to examine what you think you know about an object. That object could be a computer, but in this example it will be something more ancient. Because we may know many things about an object, but we are trained to think about everything in a limited number of ways. This isn't how the brain works and it stifles our learning when we do this. So let's practice with a very simple artifact of human technology and a symbol common in mythology and ritual-- a dagger.

If you hold out your hand and place a dagger on it, (yes, with the sharp parts not touching the skin so that it causes any sort of injury) it is just a piece of metal. You wouldn't want to leave it laying around where kids could get it, because it's sharp and they could get hurt. So maybe your first thought is about safety. That's fine. We have evolved for many countless years to consider things like safety-- that's very natural.

But beyond safety concerns, the object has great cultural significance. Show it to people, and most will think of war or a person attacking someone else. They probably won't think of medicine, unless you show them a scalpel designed specifically for it. They probably won't think of a dagger as an invention for healing as well as harm, even if the invention of scalpel follows naturally from the invention of the dagger.

They might not think of their dinner knife, even if it is made of the same materials and concepts that the dagger is. If they're vegetarian, they might be even less likely to think of the dagger as a dinner knife, even if they use knives in the kitchen to prepare vegetarian meals. A dagger is a dagger, and daggers hurt people. That's all. They don't lead to the invention of life-saving surgery, they don't enable hunters to avoid starving, they are certainly of no use to a vegetarian.

And this is the way schools teach people to think-- in very narrow, fact-oriented ways. Only as an exception are people taught to expand their knowledge. They are instead told to consume it, digest it, until they feel bloated and feel like they never want to eat again.

Most people don't understand computers, and sadly this includes most teachers. They are taught that computers run apps, and apps are computing. So teaching computers means training people to use apps. That's computing, so that's what computing education is like.

And that's pure mythology. Using applications designed to be just applications will not teach you anything about computers, but that's how we teach them, and that's why most people don't understand computers. If we want people to understand computers, then we have to stop teaching them the mythology that apps are computing.

Apps are the surface of computing, and an aspect of computing that requires very little of (by design) and limits (by nature of the design) an understanding of computing. Teaching computing via learning applications is like teaching cooking via going to a restaurant. And it's a fine place to start, but you won't get very far unless you step into the kitchens.

We can go further with the metaphor, too. A program is a recipe, all food is made of molecules but most cooking is simpler than molecular

physics and chemistry-- all computing is really just processing numbers, but most programming is simpler than the math that underlies it. You could, both theoretically and in practice, write an entire operating system using only numbers. Most people wouldn't do that. The history of computing from the 1950s to the present is largely about moving past that-- although algorithms are very mathematical.

Dreams and hypnosis and mythology helped Freud and Jung understand the mind, and mythology can help you understand the computer. But you have to learn naturally if you want the best chance of success-- you have to explore the ideas closer to yours, and connect ideas together to learn, rather than just recite facts.

Knowledge is a process of exploration and integration.

As for math, the only math I ask of you is addition, subtraction, division and multiplication. You can ask the computer to do the calculations for you, but you must at least understand addition is adding numbers together, and subtraction is removing one number from another.

Do you know how to multiply? If you can add, then yes. Multiplying is adding a number, one or more times.

Do you know how to divide? If you can subtract, then yes. Division is subtracting a number, one or more times.

If you know how to use a calculator, then you know how to write a program. Here is how you use a calculator to add 2 to 2; you just press the following 4 keys:

2
+
2
=

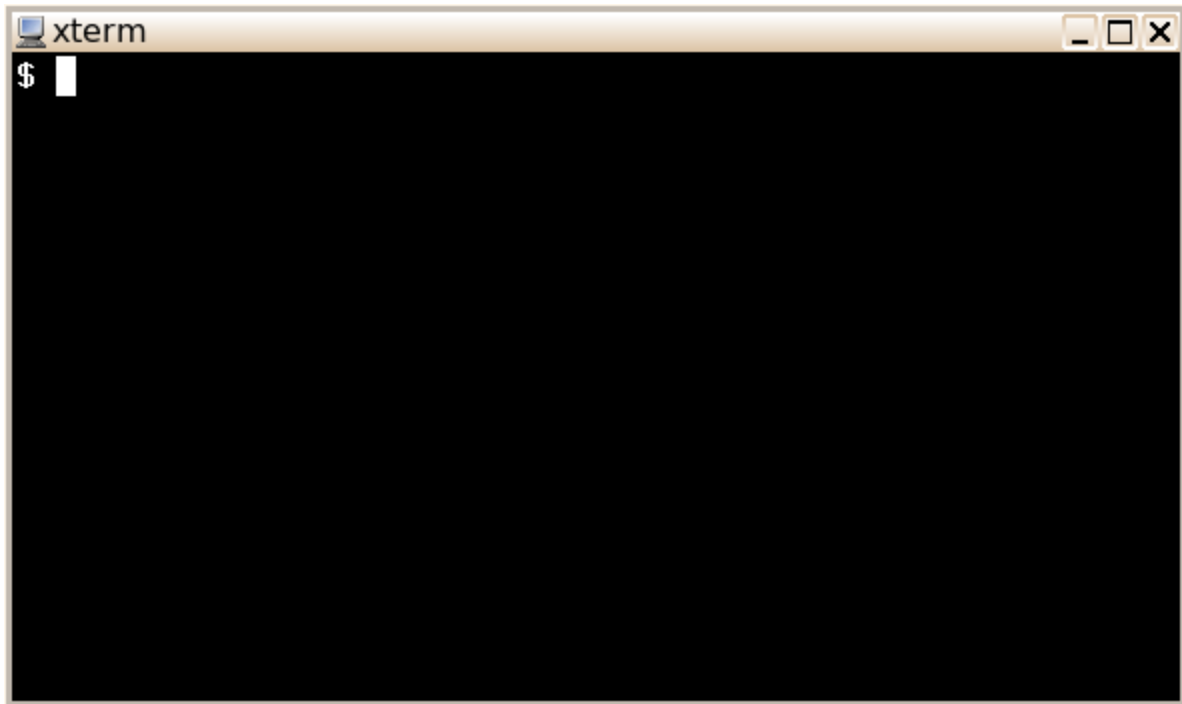
And it gives you the sum. So our "program" for adding two numbers is:

$2 + 2 =$

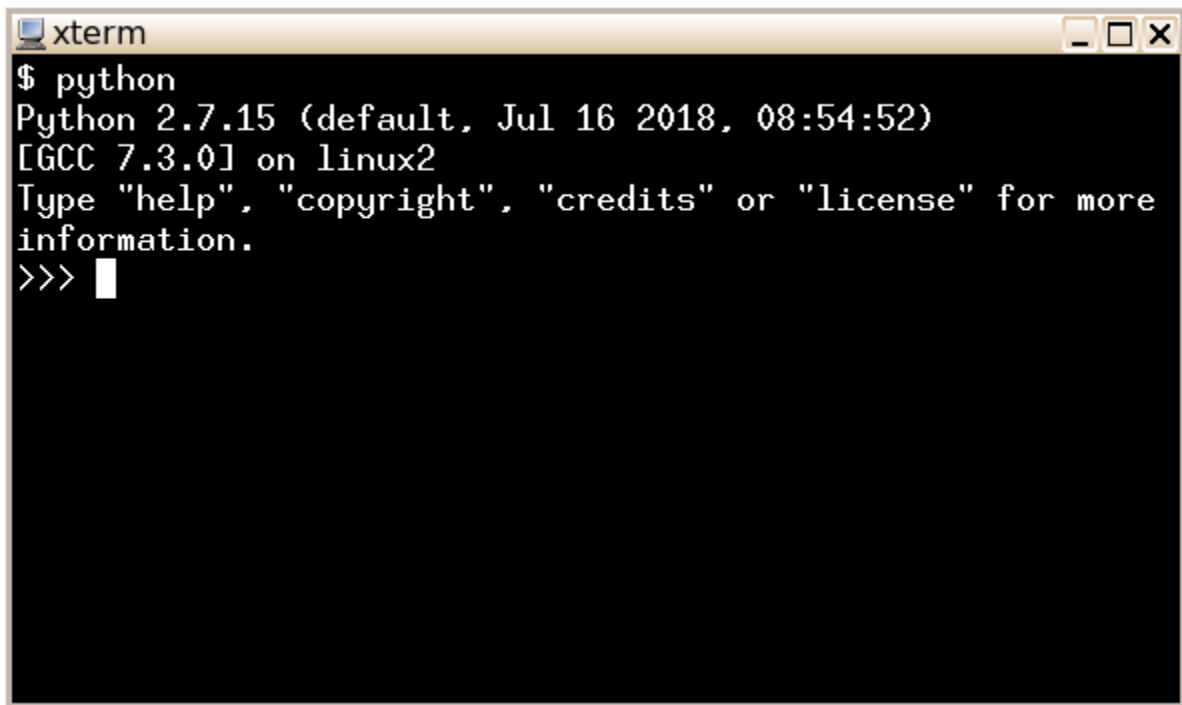
We can run the same program with words:

"Ant" + "eater" =

You may not believe this, but I will open a python session and show you myself. First I hold down CTRL and ALT on the keyboard, and hit T. That opens this window:



We have left the dining area and entered the kitchen. Now let's type "python" and get the knives and ingredients out. What are our ingredients? Data!



```
xterm
$ python
Python 2.7.15 (default, Jul 16 2018, 08:54:52)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> █
```

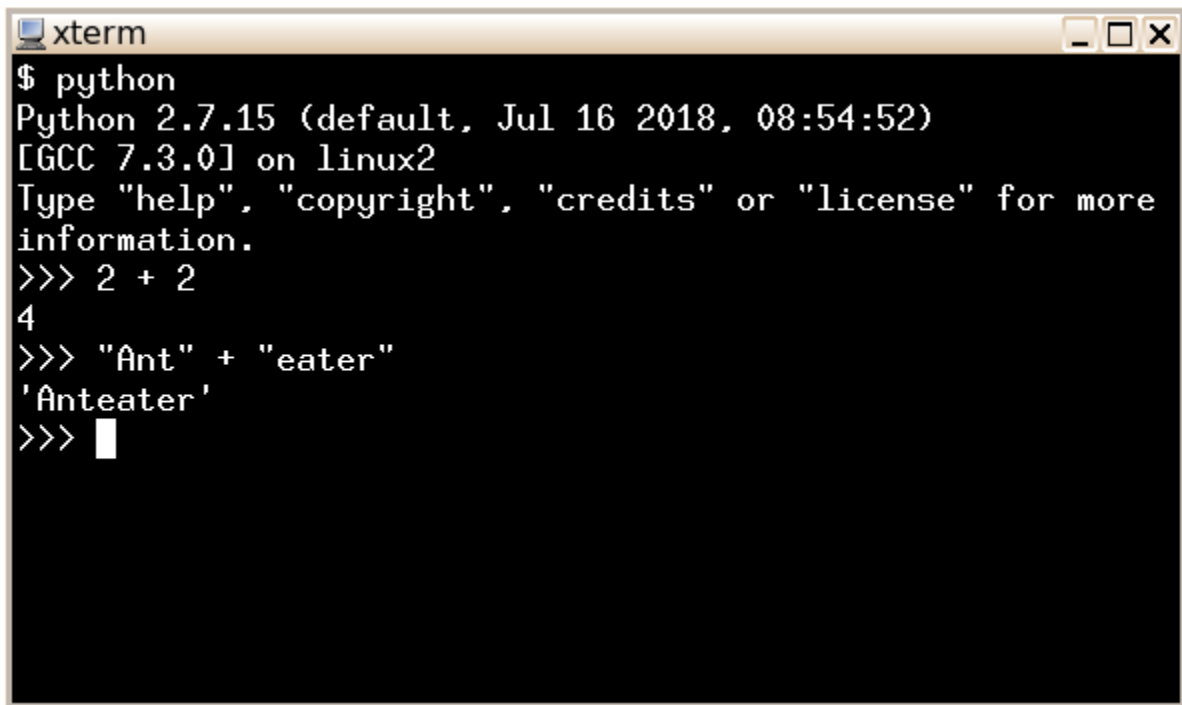
Typing "python" runs the program located at /usr/bin/python.

Can we say what / is? If you like, it's the filing cabinet. Or it's the room the filing cabinet is in. For the moment, it can be the kitchen. Maybe /usr is a particular drawer then. In the filing cabinet or the kitchen.

Maybe you like geography more than cooking. Ok, so / is the world, and /usr is a continent. So /usr/bin is a country, and /usr/bin/python is a famous french recipe. It's actually based on a recipe from the Netherlands, very literally, and the chef is Dutch.

As you learn about computers, I encourage you to treat it like a walk. You can try to memorise every feature of the journey, and the farther you walk the more that method will leave you feeling inadequate. You can write down every single thing that you see, and while you may feel more accomplished you will still find your walk very tiring.

You can also enjoy the scenery, but think about how these things might apply to other ideas. Don't worry about proof-- proof comes later, unless you want to download Python and try this too:

A screenshot of an xterm window with a black background and white text. The window title is 'xterm'. The text inside shows a terminal session where the user has run 'python'. The output shows 'Python 2.7.15 (default, Jul 16 2018, 08:54:52)' and '[GCC 7.3.0] on linux2'. It then prompts the user to type 'help', 'copyright', 'credits', or 'license' for more information. The user has entered '>>> 2 + 2' and the output is '4'. The user has entered '>>> "Ant" + "eater"' and the output is ''Anteater''. The prompt '>>>' is followed by a cursor.

```
$ python
Python 2.7.15 (default, Jul 16 2018, 08:54:52)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> 2 + 2
4
>>> "Ant" + "eater"
'Anteater'
>>> █
```

So there you go, $2 + 2$ is 4, and "Ant" + "eater" is 'Anteater'.

Don't worry about the quotes, we can avoid them when we need to.

We did use the Enter key on our giant calculator, instead of the = key on the little calculator. But the "program" is the same:

$2 + 2$ Enter

"Ant" + "Eater" Enter

The `>>>` is called a prompt. It tells you that the computer is waiting for information to be entered. The `>>>` in particular suggests that Python is waiting for information. Before we typed python, we saw a different prompt:

\$

Believe it or not, this has meaning. To an American it means money, to a UNIX administrator it means "regular user mode." Do you have to know that? No. But if you find a prompt like this:

#

It means "super user" and you can do a lot more (for better or worse) as a super user than a regular user.

You need to be the super user to write or access certain files. This is based on UNIX, a system designed in the 1970s for multiple users of the same computer-- connected over phone lines. So as user, you have access to the files in your own account.

Super user has access to all the accounts, and can create new accounts. Do you have to know this? Not to learn how to code.

In 12 pages or less, we have taken you on a quick exploration through history, underneath the application and into a programming language itself. But most importantly, we have talked about alternatives to school learning that will help you learn about computers if you stay interested. We have addressed some of the mythology that keeps people from learning about computers and other subjects.

Here's what we haven't taught you yet-- we haven't taught you how incredibly and fundamentally customisable this all is.

We haven't shown you a good concrete example of abstraction yet.

Computers are fundamentally calculators, but they abstract everything. Language is an abstraction, too. For example, the letter "A" is part of a word, it is also a number. We have used "a" in the word "part" and twice in the same sentence to mean the number 1:

"A" is p**a**rt of **a** word, it is also **a** number."

We don't know what "a" originally meant-- we know it came to English via the Phoenician alphabet, and was inspired by the Egyptians. A is most likely upside down, and the bottom "legs" were probably horns. "A" is drawn like an animal head.

If you didn't know the history of the letter, it hasn't stopped you from using the letter in words for most of your life. But as you walk through knowledge, you pick up things that are interesting to you. You can do this with computers. Computers create abstractions around numbers the same way language creates abstractions around ideas.

Through communications standards, (necessary for pieces of equipment to talk to each other) we have given the upper case "A" the number "65". This continues through the Alphabet, with "B" as 66 and so on. A space (like from the spacebar) is 32. If you add 32 to "A" you get "a".

That's from the 20th century ASCII standard, and it is incorporated into the modern Unicode standard-- the first 128 positions in Unicode (from 0 to 127) correspond to the ASCII standard that preceded it.

Do you have to know that to write programs? No, but the more you pick up along the way, the more you get a picture of how the computer works.

Everything in the computer has a number. The glowing dot in the upper left-hand corner has a number, it's 0. If your screen has 1366 columns of dots and 768 rows, that's just over a million dots-- 1049088 exactly-- and all we had to do is ask the computer to multiply 1366 by 768.

You can refer to a dot on the screen with a number between 0 and 1049087 then. But it's more reasonable to refer to it as an X value between 0 and 1365, and a Y value between 0 and 767. The computer doesn't care which way is easier for you-- it has 1049088 dots to contend with one way or the other.

Someone makes a graphics adapter, then they give it some electronics on the side that let it talk to the computer and vice-versa. But ultimately the computer is saying "Light up the dot here." And "here" is really a number.

So one programmer says "if the screen is 1366 dots across, and 768 down, and we want to light the 2nd dot on the 3rd row, that's $1366 + 1366 + 2$ -- that's the 2734th dot on the screen. But the first dot is dot 9, so the 2734th dot is dot 2733."

And they worry about the fact that it's 2733, because all the computer cares about is it's dot 2733. But you don't have to worry about that. To you, it's the 2nd dot on the 3rd row, or the dot at (1, 2) which is the same thing.

But we didn't even worry about that to make a letter "A", even though it's 65, we just said:

"Ant" + "eater"

And it figured out where to put the dots for us.

But here is a fundamental understanding of what's happening:

1. you press a key
2. that sends a number representing that key to the computer
3. some instructions look up that number and return a representation of a shape composed as dots
4. the dots are fed into a routine that says "draw a dot here"
5. those "draw a dot here" instructions are fed to the graphics hardware
6. the graphics hardware puts the dots on the screen.
7. bonus: the colours are also numbers!

How much of this do you have to know to write programs? Technically, NONE OF IT!

And the reason is abstraction. Because "coding" can be about as simple as saying:

"PUT A WHITE DOT AT 1, 2"

...or, "MAKE THE 2734TH DOT COLOR 0"

...or even, "WRITE THE WORD: ANTEATER TO THE SCREEN"

It is more common to say things like:

PSET(1, 2), RGB(255, 255, 255)

or...

PSET(1, 2), 0

or...

PRINT "ANTEATER"

Partly because there's less typing. But that's not computer language, that's language people constructed using a computer language; to the

computer, that's just a bunch of numbers telling it to send other numbers to devices.

And just like foreign languages deal with verbs and nouns and pronouns, computer languages deal with commands and data and variable names.

Because of abstraction, you can develop your own language for programming. Or learn someone else's, or both.

The fun part is, when they make computer languages that can do just about anything, but also when they make friendly introductory languages that kids can learn, there are a similar handful of concepts to getting all tasks done:

1. variables-- associating a name with some data.
2. input-- getting data from outside (sometimes outside the computer, sometimes from a storage device.)
3. output-- getting data to outside (sometimes outside the computer, or to a storage device.)
4. basic math-- really just adding and subtracting.
5. loops-- doing something repeatedly
6. conditionals-- doing something only when criteria based on comparing or measuring one or more pieces of data are met
7. functions-- abstraction; small collections of other instructions that you can build or alter to create your own programming language (or operating system.)

You can teach yourself ALL of this. But with a good teacher, who joins you on your exploration through knowledge rather than teaching you to study only the surface-- can help you learn more easily, more efficiently, and faster.

- **license: creative commons cc0 1.0 (public domain)**
- <https://creativecommons.org/publicdomain/zero/1.0/>